UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Aivar Lobjakas

Hardware-in-the-loop testing module for Mission Control System

Bachelor's Thesis (9 ECTS)

Supervisors:

Kaarel Hanson

Indrek Sünter

Mare Koit

Tartu 2016

Hardware-in-the-loop testing module for Mission Control System

Abstract:

CubeSats are small satellites, standardized by size and mass to reduce the development and satellite launch costs, but they are often not tested thoroughly enough, causing mission failures. The aim of this thesis is to design and implement a testing module for executing automated functional tests on any CubeSat subsystem. The testing module will also save test results and subsystem communication logs. This system would allow detailed testing of satellite components, improving the reliability of the tested satellite. The complete testing system requires the testing module software, which is the subject of this thesis, and a hardware platform for subsystem communication, that is not in the scope of this thesis. First functional version of the required software is implemented by the author. The design and implementation of the software is explained, including the integration to an existing Mission Control System, an application used for satellite and ground station control and monitoring.

Keywords:

Space technology, CubeSat, Mission Control System, ESTCube-2, testing

CERCS: P170 Computer science, numerical analysis, systems, control

Riistvara testimismoodul missioonijuhtimissüsteemile

Lühikokkuvõte:

Kuupsatelliidid on väikesed satelliidid, mis on standardiseeritud suuruse ja massi järgi, et vähendada satelliidi arenduse ja üleslennutamise kulusid, kuid tihtipeale neid ei testita piisavalt, põhjustades missioonide ebaõnnestumisi. Käesoleva töö eesmärk on disainida ja implementeerida testimismoodul, millega saaks sooritada automaatset funktsionaalset testimist iga kuupsatelliidi alamsüsteemil. Testimismoodul salvestab ka testitulemusi ja alamsüsteemiga suhtluse logi. See süsteem aitaks detailsemalt testida satelliidi komponente, parandades testitava satelliidi töökindlust. Täielik testimise süsteem vajab tarkvaralist testimismoodulit, mis on käesoleva töö teema, ning alamsüsteemiga suhtluseks vajaminevat riistvaraplatvormi, mis käesoleva töö juurde ei kuulu. Esimene funktsioneeriv versioon vajaminevast tarkvarast on autori poolt implementeeritud. Selgitatakse tarkvara disaini ja implementatsiooni, muuhulgas integreerimist olemasoleva missioonijuhtimissüsteemiga, mis on satelliidi ja maajaama juhtimiseks ning jälgimiseks mõeldud rakendus.

Võtmesõnad:

Kosmosetehnoloogia, kuupsatelliit, missioonijuhtimissüsteem, ESTCube-2, testimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

High cost of traditional space missions and the continued miniaturization of commercial electronic devices has increased the number of nano-satellite missions. The increasing capability of consumer electronics means that the capability of nano-satellites will increase as well, improving the cost-effectiveness of small satellites [1].

CubeSats are a type of nano-satellites, that are standardized by size and mass, which creates a cost-effective way of performing technology demonstrations or science experiments in space [2]. Relatively low cost means that CubeSats can be used to test experimental equipment in space, which otherwise would be too expensive to create a traditional satellite for. For example, the mission of CubeSat ESTCube-1 was to test an innovative electric solar wind sail, "that could revolutionize transportation within the solar system" [3]. The Aalto University satellite Aalto-1 also has the same electric solar sail mission [4]. Another example would be SwissCube, which had a scientific mission to photograph air-glow [5].

However, the reliability of CubeSats is a problem. The failure rate of university-led CubeSat missions is very high, with nearly 50% of the satellites failing to meet their mission objectives. Most of the failures could be avoided with more thorough system-level testing [6]. Having an automated platform for both component- and system-level testing would help to improve the reliability of CubeSats greatly.

At the moment there are no suitable platforms available to use in the ESTCube program for testing each subsystem independently, without the use of a broker, such as the satellite's onboard computer. Independent testing would allow to save time, because one or more subsystems might be ready for the testing phase, while the onboard computer might not. Currently, this would mean that testing of the subsystems that are ready, can not be started until the onboard computer is ready for the testing phase as well. That is one of the major issues in adopting an Agile methodology for embedded hardware and software development.

Because there was no platform for testing the satellite subsystems for ESTCube-1, custom software was developed for testing, that was not part of the existing Mission Control System (MCS). This meant there were two separate systems in development and due to time constraints, the testing software was not a universal solution and did not satisfy all the requirements.

The overall solution for creating a new testing platform for the satellite subsystems depends on another Bachelor's Thesis, the aim of which is to develop a hardware platform for connecting satellite subsystems or their components to the testing module, that is created in the scope of this thesis. The hardware platform performs the translation to USB from SPI, I2C, RS485 or any other low-level protocols. In addition, the hardware platform would enable the injection of errors in the data stream, to see if and how the subsystem would recover from it.

The aim of this thesis is to make the testing of hardware more systematic and manageable for the ESTCube missions, by creating a testing module that can be used to test each of the subsystems of the satellite independently. It is planned to first use this testing module for ESTCube-2. The testing module will be part of the existing MCS, thus creating a single system, as opposed to two separate systems that existed for ESTCube-1. It will also be integrated with the scripting engine module of the MCS. The testing module will allow to automate satellite subsystem testing and save the test results. For debugging purposes, the logs of the commands sent to the subsystem as well as the responses received are also saved. The scope of this thesis consists of designing the testing module and implementing a first functional version of it. This includes getting familiar with the MCS code base, the scripting engine in

particular, integrating the testing module into the MCS, requirements and use case analysis, and functional testing of the developed system. The scope of this thesis does not include developing the hardware platform for testing module and subsystem communication nor the protocol needed for this communication.

Chapter 2 of the thesis gives background information about the design of CubeSats, missions of ESTCube-1 and 2 and the design of MCS, including information about the scripting module and Groovy scripts. Description of the problem at hand is also given in the 2nd chapter. Chapter 3 contains the requirements for the testing module. Chapter 4 describes the design and chapter 5 includes the implementation details of the testing module. Chapter 6 describes an example test, that was run on the testing module. Finally, Chapter 7 discusses the future development plans of the system.

Appendix A contains a small sample of ESTCube-1 contact times. Appendix B includes a Groovy test code, that is used for demonstrating the usage of the testing module. Appendix C contains code that is run on the Arduino microcontroller when executing the test from Appendix B. Appendix D contains the URL for the developed testing module repository. Finally, Appendix E is for the non-exclusive licence to reproduce thesis and make thesis public.

2 Background

2.1 CubeSat design

CubeSats are nano-satellites, that are standardized to reduce the development time and the cost of the satellite [7]. A CubeSat is a cube with an approximate volume of one liter and with a mass up to 1.33 kg [8]. The CubeSat standard allows for bigger CubeSats, that are 1.5, 2 or 3 times larger in height and can weigh more [9]. CubeSats consist of multiple subsystems, the subsystems and construction of ESTCube-1 can be seen on Figure 1.

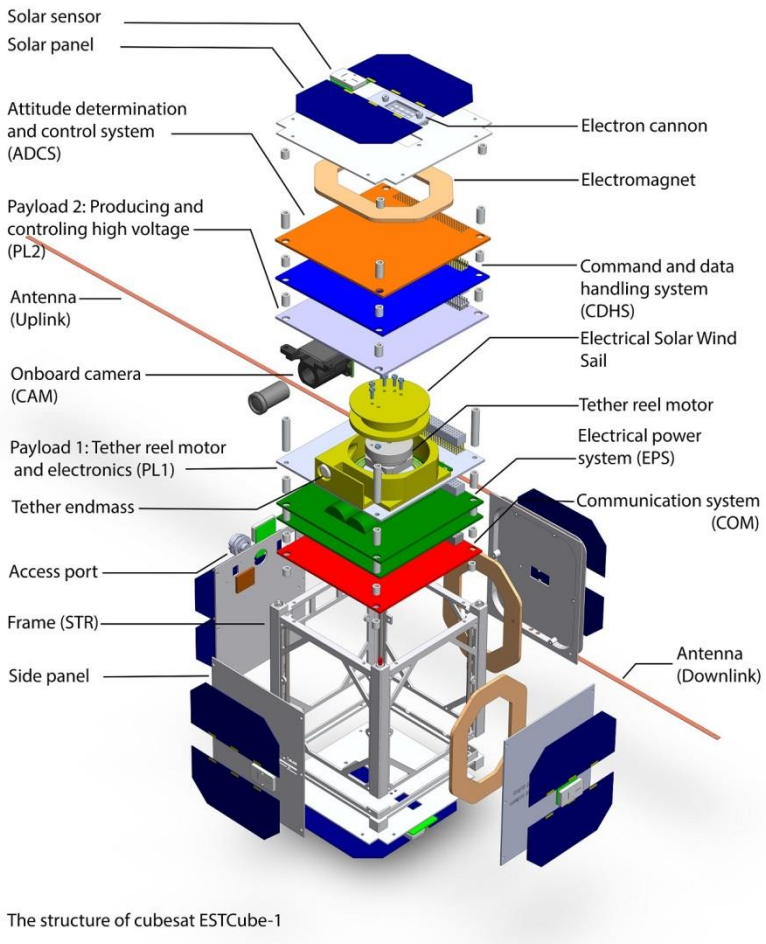

Figure 1: ESTCube-1 structure [10]

2.2 ESTCube-1 and ESTCube-2

ESTCube-1 was a student satellite project of the University of Tartu, that started in 2008. The main objectives included getting students to participate in space projects and perform a test of the electric solar wind sail (E-Sail) [11].

ESTCube-2 is the second satellite project of the University of Tartu and its objective is to test the Coulomb drag propulsion and advanced satellite subsystem solutions [12].

To test the Coulomb drag propulsion, the satellite will have a 300 m tether on board, that will be deployed and charged to decrease the satellite's altitude. It is estimated that this tether „could deorbit ESTCube-2 from the altitude of 700 km to 500 km in half a year." This technology could then be used to decrease the amount of space debris [13].

The satellite will contain the following subsystems: „electrical power system, communication system, on-board computer, and attitude control system". The satellite will be built to minimise mass and volume, so that it could be used as a platform for future missions [12].

2.3 Mission Control System (MCS)

The MCS is an open-source web-based software, developed for ESTCube-1, that is used for real-time satellite communication and tracking and also for monitoring and controlling ground stations and their components. It is the first web-based and open-source mission control system. This means that it is possible to have ground stations all over the world for more frequent satellite communication and it can also be used by other satellite projects [14]. In addition, a web-based mission control system enables satellite operations from anywhere. Traditional spacecrafts are typically operated from a specific mission control room. With students attending lectures and practicals, this would not be applicable.

2.3.1 MCS design

The component-level design of the MCS can be seen on Figure 2. The MCS uses HummingBird, an open-source "monitoring and control framework for small satellites" [15], for many of its main functions, including satellite tracking and the prediction of contact times. The Scripting block on Figure 2 refers to the scripting engine, that has not yet been used to operate a satellite.

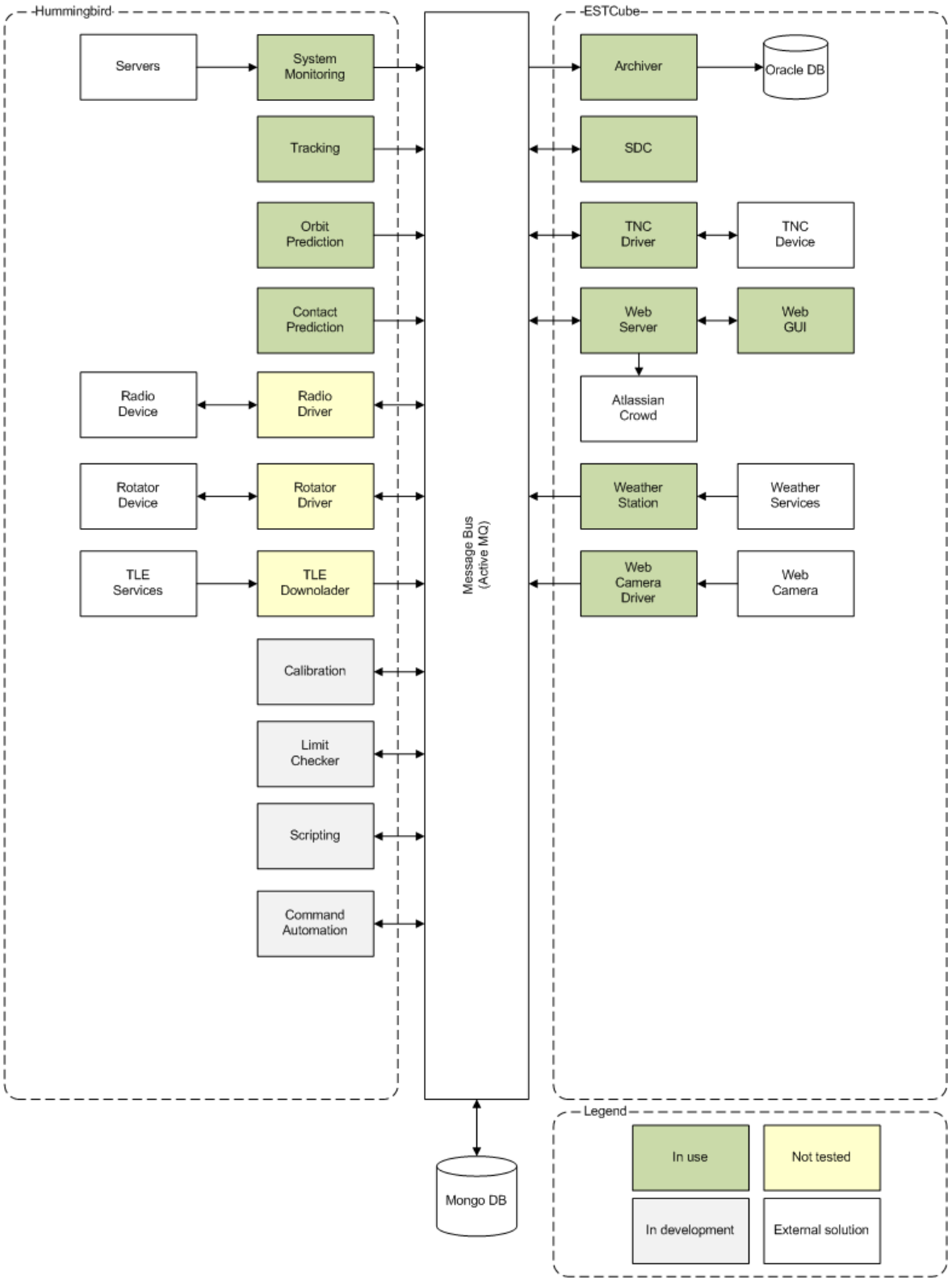

Figure 2: MCS design [16]

2.3.2 Scripting engine

The testing module, that will be created, will use the existing scripting engine module of the Mission Control System. The scripting module was created to help the automation of the satellite passes using Groovy scripts. Automating the satellite communication is required to maximize the available communication time with the satellite. For ESTCube-1, the contact time with the satellite was about 5 to 13 minutes, depending on the trajectory of the satellite in relation to the ground station, with approximately 90 minutes between contacts during the day and multiple hours during the night. Some example contact times are in Appendix A. The scripting module uses the Groovy version 2.4.3.

The scripting engine is responsible for compiling the Groovy script and executing it. It also adds extra methods *send* and *listen* to Groovy, that allow to send data to the satellite and listen to the responses. The separate *listen* method is necessary, because the commands, that are sent, may not have useful responses in the given context or any responses at all. As the responses must arrive within a specific timeframe, the scripting engine will throw an error, if that time limit is exceeded.

Having the scripting module in the MCS means that, before the next contact with the satellite, the satellite operator can create a script, that can execute the necessary commands and choose the next actions based on the incoming responses. That means more communication time with the satellite, because the operator does not have to manually enter the commands and analyze the responses, to figure out what the next command should be.

2.3.2.1 Apache Groovy scripts

Groovy is a dynamic programming language, with a concise syntax, for the Java platform, which allows it to integrate and interoperate with Java and any Java third-party libraries. It also has scripting capabilities and customization mechanisms, for creating Domain-Specific Languages [17].

Groovy scripts are compiled to Groovy classes, where the main body of the script will be copied to a *run* method, which will be executed in the *main* method of the generated class [18].

2.4 Problem

Testing the satellite subsystems for ESTCube-1 had quite a few problems, many caused by the lack of time. Automated testing was tried using Elvior TestCast [19], a platform for developing and executing tests, but the work overhead of developing tests in an unfamiliar programming language was not worth the time. But there is a strong need for automated testing, because minor changes to subsystem hardware or firmware may cause major regressions that could be overlooked during manual functional test runs. Also due to time constraints, a short description of what tests were done was given, instead of detailed test reports, which means there was no clear overview of the testing process. Not having a clear overview can possibly cause a lower test coverage of the equipment under test (EUT) or cause unnecessary repetition of a test, wasting valuable development time. In addition there is a need for testing individual sensors or actuators without dedicated hardware, which was not possible during the testing of ESTCube-1.

3 Requirements

The requirements, that the finished testing module must fulfill can be separated into the following main groups: functionality and user interface; data logging; portability.

3.1 Functionality and user interface

Requirements for functionality and user interface:

1. User must see which tests have passed (green) and which have failed (red).
2. User must see the script debug output and commands sent to the hardware and responses received.
3. User must be able to insert commands.
4. User must be able to view the command history.
5. User must be able to see:
    a. script name that's currently running;
    b. if the script is running;
    c. if there's a connection to the hardware.
6. User must be able to start, stop, and pause tests and also execute tests one step at a time.
7. User must be able to load and run previously written tests.

3.2 Data logging

Requirements for logging data:

1. Output of script debug must be logged to a file.
2. Test verification results must be logged to a file.
3. Communication between the hardware must be logged to a file.
4. All executed commands must be logged to a file.
5. The rotation time for the logs must be configurable.

3.3 Portability

Requirements for portability:

1. Software needs to be able to run on Linux, Windows and Mac.
2. Software running on the server must be accessible over the internet.

4 Design

The desired testing solution can be separated into two separate parts, software and hardware. The software is the testing module, which is part of the MCS and the hardware is the adapter that will be used to connect the satellite subsystems to a computer that runs the testing module. This thesis concentrates on the software solution and is not responsible for the hardware. Unfortunately the hardware adapter was not completed during the writing of this thesis and thus the desired end product was not completed. Plans to finish the hardware are underway, so that the testing of ESTCube-2 subsystems could be started.

4.1 Design overview of the desired solution

The MCS executes the script that the user entered and if the script contains a send method, it passes the method argument over a Transmission Control Protocol (TCP) [20], connection to the testing module. The TCP connection is used due to future considerations, where the testing module will be separated from the MCS, so that the hardware that needs to be tested would not need to be connected to the server that runs the MCS.

The testing module then relays the command over a Universal Serial Bus (USB) connection to the adapter. The commands can be addressed directly to the adapter, for example there could be a command to enable the monitoring of some specific voltages or currents on the adapter module. In that case, the adapter executes the command, otherwise it sends the command to the subsystem, that is currently being tested. If either the subsystem or the adapter produces any responses to the command, then they are sent back to the testing module.

The responses, any exceptions that may have occurred, and notifications, are sent from the testing module back to the MCS. For example, the notifications include messages, that contain information about whether a connection was successfully opened or not. The connection can be either a connection from the testing module to the adapter, from the adapter to one of its modules or from one of its modules to the EUT.

The design of the desired solution is illustrated on Figure 3.

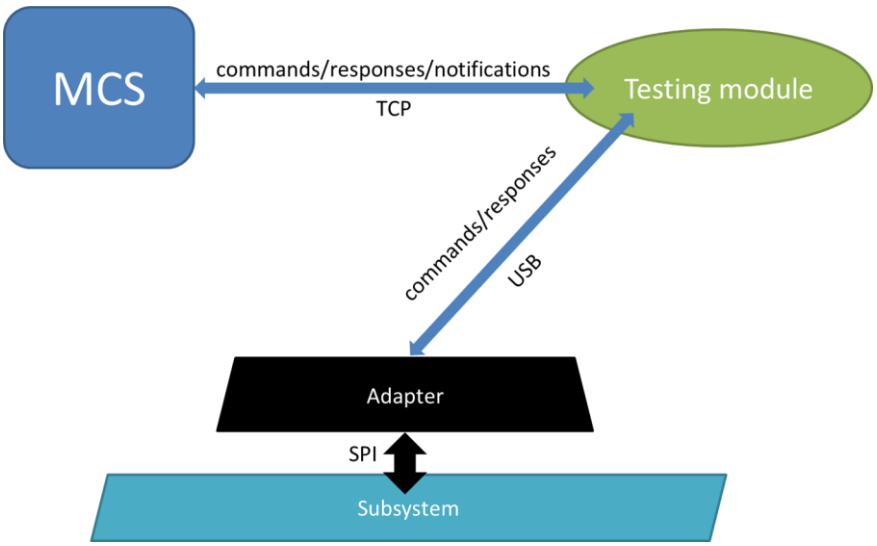

Figure 3: Design overview of the complete testing system solution

4.2 Design overview of the current system

At the time of writing this thesis, the development of the adapter has not been completed. In order to fulfill the requirement 2.1 in chapter 3, which states that the communication between the hardware must be logged to a file, a hardware component is needed that could simulate the adapter and the subsystem. An Arduino microcontroller is used for this purpose. The adapter is simulated by the USB connection port on the microcontroller and the subsystem is simulated by a program uploaded to the microcontroller, that would generate responses to commands. This means that it is necessary to write a simple program for the Arduino, that would listen to the incoming data and generate a dummy response to send back to the testing module.

The design of the current solution is illustrated on Figure 4.

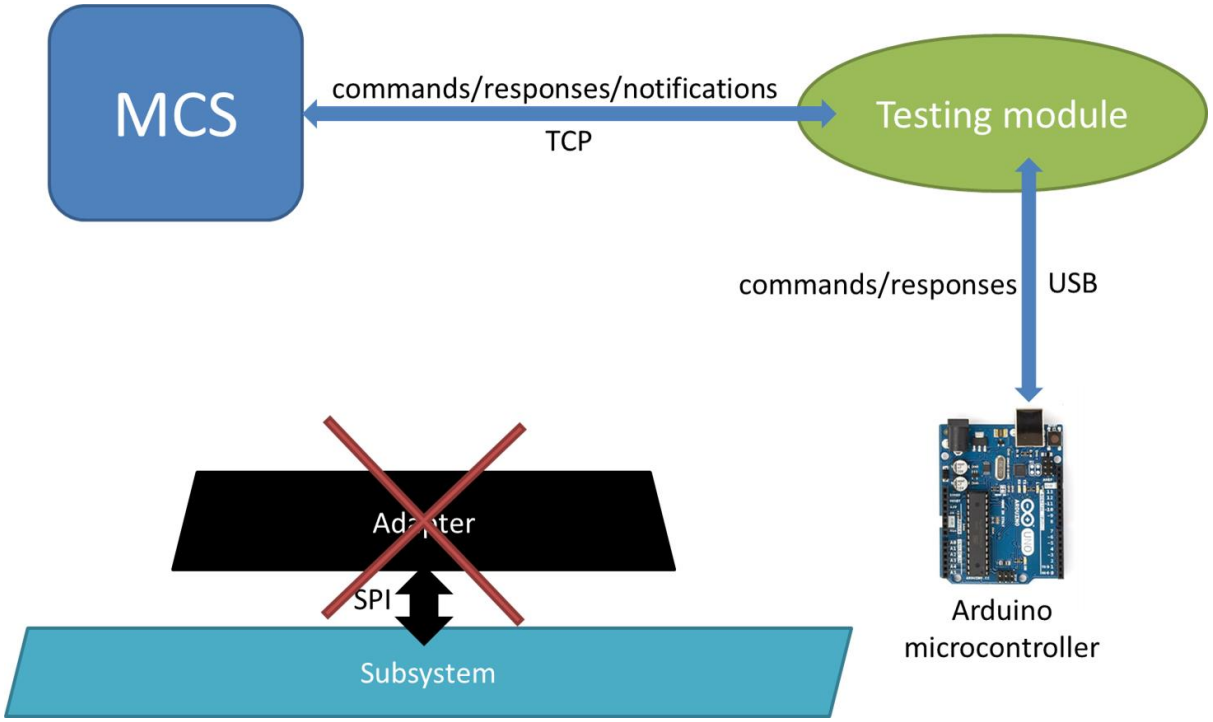

Figure 4: Design overview of the current testing system solution [21]

4.3 Data flow

Figure 5 illustrates the data flow throughout the subsystem testing process. It also shows the development that has been done within this thesis' scope, with the light green boxes representing the modified parts of the software and the dark green boxes representing the software that did not exist before.

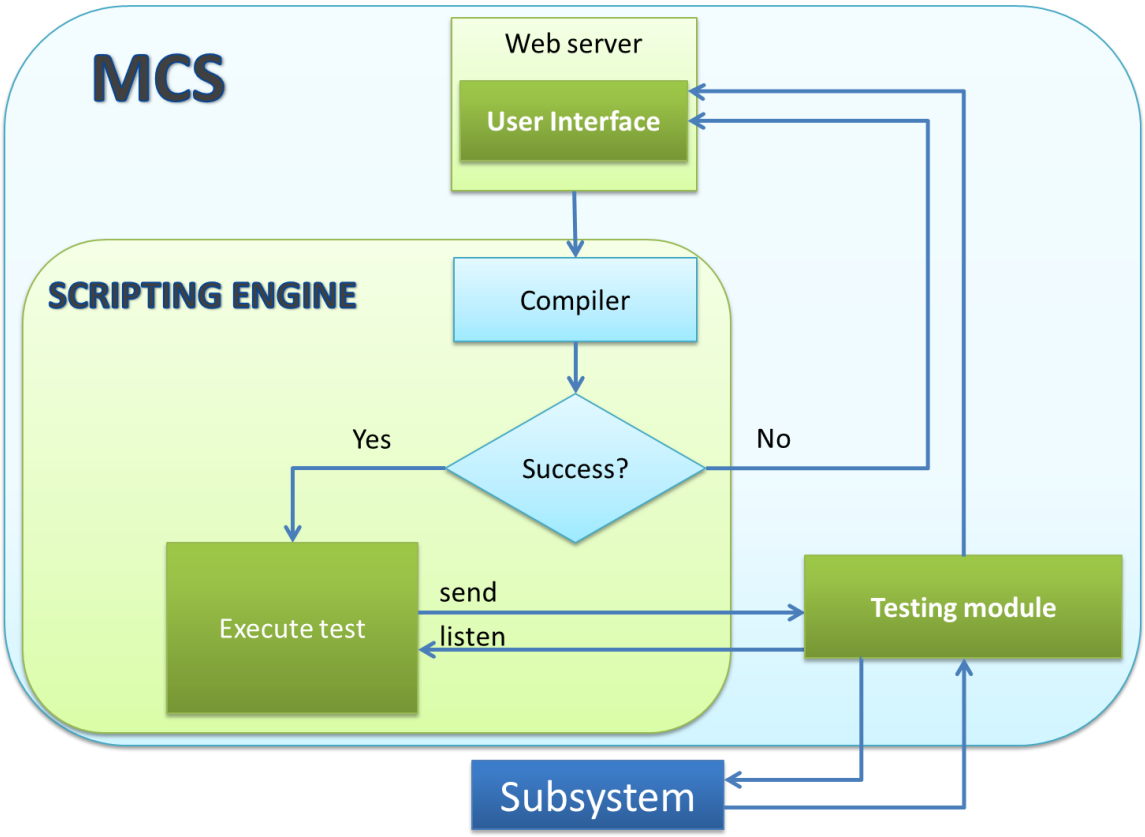

Figure 5: Data flow chart

The user enters a Groovy script into the user interface. When the script is started, it is first sent to a compiler, which is part of the scripting engine. If the compilation fails, then the exceptions, that are thrown, are sent back to the user interface, so the user could fix the error in the script, that caused the compilation failure. If the compilation is successful, the script will be executed and the methods *send* and *listen* are used to communicate with the testing module. The testing module will send the commands to the subsystem and gathers any responses coming from the subsystem. If the script needs the response, it will have to use the *listen* method, which will signal the testing module to send the subsystem's response back to the scripting engine.

Listening to a response will block the execution of the rest of the script, because different responses might affect the rest of the script execution. For example, asking for and then listening to the satellite's current attitude, could be used to decide, what commands should be executed next, using Groovy's control flow statements. The listen command will not block indefinitely though, because the command may not have reached the subsystem or an error in the subsystem causes it to not respond at all. After a certain amount of time, the *listen* method will throw a timed out exception, which will stop the script execution.

4.4 Groovy tests

The construction of the Groovy tests written for the testing module can be seen in Appendix B. Separate tests can be created by using *states*, which will internally be compiled to a Groovy class by the scripting engine. The scripting engine is modified to catch any assertion errors, which are used to decide whether a test has failed or passed. If a test fails, then the line number, that caused the failure will be displayed to the user in the test results, along with the description of the cause. Example of this can be seen on Figure 16.

5 Implementation

5.1 User interface

The MCS uses Bootstrap for creating the HTML of the user interface and AngularJS for JavaScript, so the testing module is created using the same frameworks.

The user interface consists of the following main components:

1. Log window: displays the notifications and hardware communication.
2. Script window: the user edits a previously written script or creates a new one. The script window is created using Ace editor, which is a code editor for the web, that „matches the features and performance of native editors such as Sublime, Vim and TextMate" [22].
3. Script load and control window: the user can load a previously created script into the script window and run the script.
4. Hardware connection window: the user can choose the USB port, that the hardware is connected to, connect to it and also disconnect.
5. Test results window: shows all of the tests, that the script contains, the result, and if the test fails, it will show the line number, that caused the failure and also the cause of the failure. Below the test results is a button for saving the test results to a file.

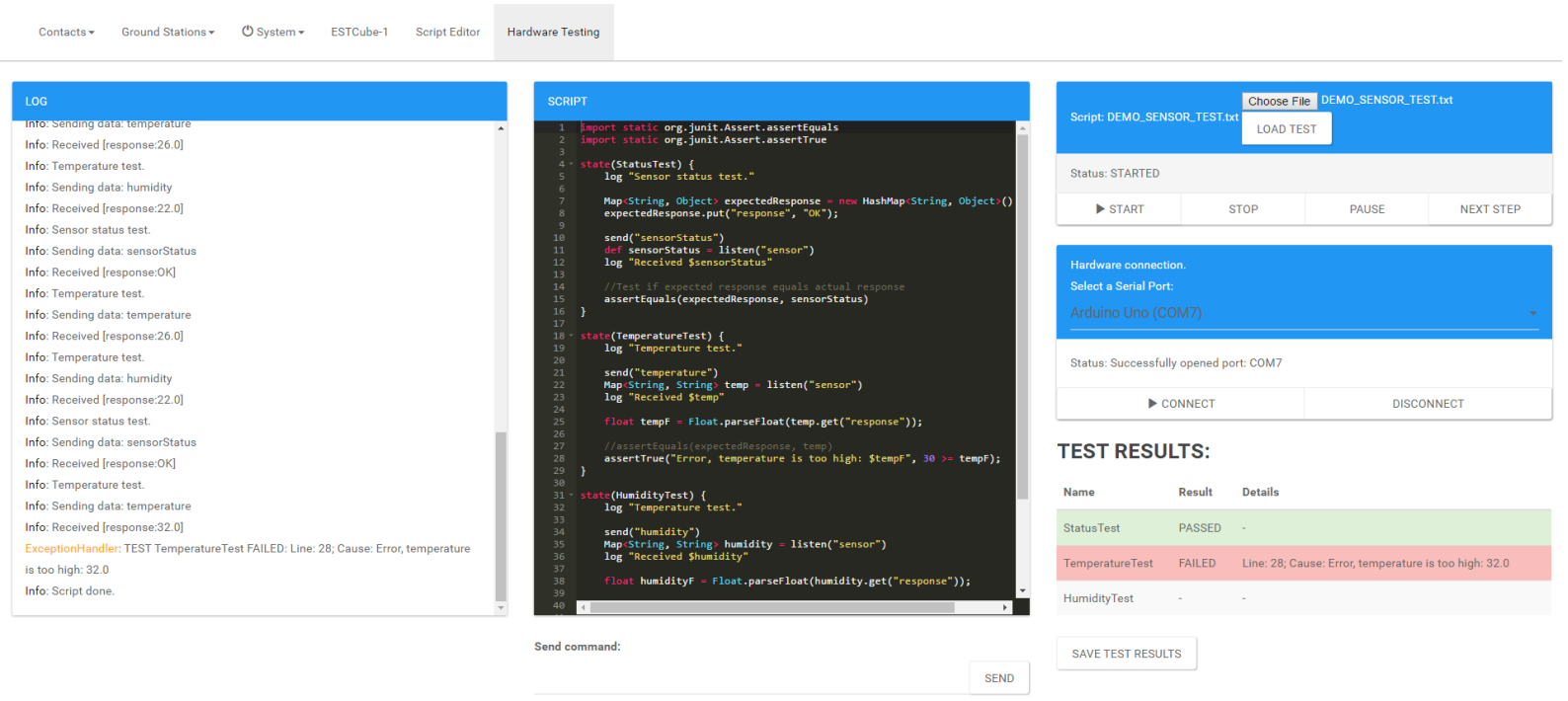

Figure 6: User interface of the testing module

The data sent to the server from the user interface is the USB port connection information and the Groovy testing script. This is achieved using HTTP POST requests.

The data sent from the server to the user interface is the script log, connection information and test results. This communication uses websockets [23], for event-driven communication. This helps to keep the architecture of the system more homogeneous, because the server uses Apache Camel routes

for communication, which is also event-driven. The advantage of using AngularJS is that it uses two-way data binding, which means that changes to the JavaScript objects are immediately shown in the HTML [24]. This is useful when using websockets for communication, because listening to websocket events and changing the JavaScript objects based on those events, means that the user also sees these changes in the user interface as soon as the events are coming in.

5.2 Integrating the testing module with the scripting engine

Thorough analysis of the scripting engine was needed before successfully integrating the testing module with the scripting engine. This section describes the steps taken to perform the integration.

5.2.1 From the user interface to the web server

The testing script is sent from the user interface, using an HTTP POST method to a HardwareTestSubmitServlet class' *doPost* method shown on Figure 7. In the method, a new Script object is created, the script code is added to the object's code field and also an identifier is added, to separate the testing script from regular scripts, that are used for satellite communication automation. The Script object is then sent to the ActiveMQ Queue, with the identifier "estcube.hardwaretesting.script". An ActiveMQ Queue is used instead of Topic, so that the message would not be lost, if there are no consumers listening to the endpoint when the message is sent [25].

```
1    @EndpointInject(uri = "activemq:queue:estcube.hardwaretesting.script")
2    ProducerTemplate testingScriptProducer;
3
4    @Override
5    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
6        StringBuffer jb = new StringBuffer();
7        String line = null;
8        try {
9            BufferedReader reader = req.getReader();
10           while ((line = reader.readLine()) != null)
11               jb.append(line);
12
13           JSONObject jsonObject = new JSONObject(new
14                   JSONTokener(jb.toString()));
15
16           String code = (String) jsonObject.get("code");
17
18           Script script = new Script();
19           script.setCode(code);
20
21           script.setIdentifier("hardwaretest");
22
23           LOG.info("Sending hardware test body to scripting engine");
24           LOG.info(code);
25           testingScriptProducer.sendBody(script);
26
27           Map<String, String> data = new HashMap<String, String>();
28
29           responseSupport.sendAsJson(resp, toJson, data);
30       } catch (Exception e) {
31           LOG.error("Failed to handle Script submit", e);
32           throw new ServletException("Failed to handle Script submit", e);
33       }
34   }
```

Figure 7: Post method of the testing script submit servlet

5.2.2 From the web server to the scripting engine

The Script object is consumed by a Camel route in the ScriptEngine class, as seen on Figure 8. The object is then processed by the TestingScriptRunProcessor class object, which implements Camel Processor interface. “The Processor interface is used to implement consumers of message exchanges” [26].

```
1   public static final String HARDWARE_SCRIPT = "activemq:queue:estcube.hardwaretesting.script";
2
3   @Autowired
4     private HardwareTestingCamelScriptIO hardwareTestingScriptIO;
5
6   /* Some code omitted for brevity. */
7
8       from(HARDWARE_SCRIPT)
9               .process(testingScriptRunProcessor)
10              .to(CamelScriptLogger.SCRIPT_MESSAGE);
```

Figure 8: Testing module’s Camel route in ScriptEngine class

The TestingScriptRunProcessor class is where the script is compiled and executed. The script is compiled to a ScriptBase object using the script engine, that internally uses the GroovyShell [27] class, which is capable of parsing and executing Groovy scripts. The ScriptBase is a custom class of the scripting engine, that adds new methods for the Groovy script to use, including *send* and *listen* methods, that are used for satellite communication. The methods, seen on Figure 9, are modified, so that the ScriptIO object would be different, depending on the script identifier. The script identifier for the testing module is set to “hardwaretesting” in the HardwareTestSubmitServlet class, which is now used to get a new ScriptIO object, created for the testing module. This is needed, because the ScriptIO is responsible for determining, where the *send* commands will be sent to and where it listens to the responses in the *listen* method. If the script’s identifier is “hardwaretesting”, then the ScriptIO object will be based on the HardwareTestingCamelScriptIO class, which is used for communication with the testing module, otherwise it will be based on the CamelScriptIO class, used for satellite communication.

```
1       public void send(String name, Map opts) {
2           ScriptIOPayload payload = new ScriptIOPayload();
3           ScriptIO scriptIO = context.getScriptIO();
4           if("hardwaretest".equals(identifier))
5               scriptIO = context.getHardwareTestingScriptIO();
6
7           if (opts != null)
8               payload.putAll(opts);
9
10          scriptIO.send(name, payload);
11      }
12
13      /* Some code omitted for brevity. */
14
15      public Map listen(String name, Map opts) {
16        int timeout = 250000; // TODO move to service.properties
17        if (opts != null) {
18            timeout = getIntOr(opts, "timeout", timeout);
19        }
20
21        final long start = System.currentTimeMillis();
22        while (true) {
23            ScriptIO scriptIO = context.getScriptIO();
24            if("hardwaretest".equals(identifier))
25                scriptIO = context.getHardwareTestingScriptIO();
26
27            ScriptIOPayload pl = scriptIO.poll(name);
28            if (pl != null)
29                return pl.getMap();
30
31            long elapsed = System.currentTimeMillis() - start;
32            if (elapsed > timeout)
33                // TODO this should probably be some other exception type
34                throw new RuntimeException("timed out");
35
36            try {
37                Thread.sleep(10);
38            } catch (InterruptedException ignored) {
39            }
40        }
41    }
```

Figure 9: ScriptBase send and listen methods

The HardwareTestingCamelScriptIO class is similar to the scripting engine's CamelScriptIO class. The key difference can be seen in Figure 10, on lines 21 and 37, that determine, where the ScriptBase's *send* command is being forwarded.

```java
@Component
public class HardwareTestingCamelScriptIO implements ScriptIO, Processor {

    public static final String TESTINGSCRIPT_OUT = "activemq:topic:estcube.hardwaretesting.command";
    public static final String TESTINGSCRIPT_IN = "activemq:topic:estcube.hardwaretesting.response";

    private final ProducerTemplate producerTemplate;

    @Autowired
    public HardwareTestingCamelScriptIO(ProducerTemplate producerTemplate) {
        this.producerTemplate = producerTemplate;
    }

    @Override
    public void send(String op, ScriptIOPayload payload) {
        ScriptCommand cmd = new ScriptCommand();
        cmd.setCommandName(op);
        cmd.setPayload(payload);

        this.producerTemplate.sendBody(TESTINGSCRIPT_OUT, cmd);
    }

    private ReplyQueue<ScriptIOPayload> incomingReplies = new ReplyQueue<ScriptIOPayload>(3000);

    @Override
    public ScriptIOPayload poll(String name) {
        return incomingReplies.poll(name);
    }

    @Override
    public void process(Exchange exchange) throws Exception {
        Message m = exchange.getIn();
        String replyName = "sensor";
        ScriptIOPayload replyPayload = new ScriptIOPayload();
        String[] keyValue = ((String) m.getBody()).split(";");
        replyPayload.put("response", keyValue[0]);

        incomingReplies.offer(replyName, replyPayload);
    }

}
```

Figure 10: HardwareTestingCamelScriptIO class

5.2.3 From the scripting engine to the testing module

Figure 11 shows the Camel routes that are communicating directly to the testing module. The commands are coming in from the HardwareTestingCamelScriptIO class and the responses are sent back to the same class for processing. As can be seen from the Figure 11, TCP is used for the communication, to make it easier to separate the testing module from the MCS in the future.

```java
// Send commands to testing module
from("activemq:topic:estcube.hardwaretesting.command")
    .to("mina2:tcp://localhost:" + "6200" + "?transferExchange=true&sync=false");
// Recieve commands from testing module
// Will be sent to [HardwareTestingCamelScriptIO]'s process(exchange) method.
from("mina2:tcp://localhost:" + "6201" + "?textline=true&sync=false")
    .to("activemq:topic:estcube.hardwaretesting.response");
```

Figure 11: WebServer class' Camel routes for testing module communication

5.3 Hardware communication in the testing module

Communication with the hardware is accomplished by two threads, one for sending data and the other for receiving responses. The threads are started after the user has selected the USB port they want to connect to. The user is shown descriptive names for the available ports, to make the selection easier. For example, if an Arduino Uno is connected to port COM7, the user interface will display that as "Arduino Uno (COM7)". For handling the serial port connectivity and communication, a platform-independent

jSerialComm [28] Java library is used. Exceptions and other useful information, including the communication with the hardware, are logged to a file using Java logging library Apache Log4j [29].

5.3.1 Sending data to a serial port

If the communication with the serial port is opened, the commands coming from the Groovy script are added to a Java BlockingQueue. The thread for sending the data takes the elements from that queue one at a time and if a command exists, the information is logged and then sent to the serial port. If there are no elements in the list, the thread blocks until new commands are available [30]. The implementation of the thread can be seen on Figure 12.

```
sendDataThread = new Thread(){
    @Override public void run() {
        System.out.println("SendDataThread started.");
        // Wait after connecting, so the bootloader can finish.
        try {Thread.sleep(5000); } catch(Exception e) {}
        while(true){
            try {
                String data = sendDataQueue.take();
                LOG.info("Sending data: " + data);
                HardwareTestingMessage htm = new HardwareTestingMessage(Type.Info, "Sending data: " + data);
                serialPortWebSocketProducer.sendBody(htm);

                //handle the data
                PrintWriter output = new PrintWriter(serialPort.getOutputStream());
                output.print(data);
                output.flush();
            } catch (InterruptedException e) {
                LOG.error("Error occurred:" + e);
            }
        }
    }
};
```

Figure 12: Thread used for sending data to a serial port

The thread for listening serial port data is using jSerialComm library's InputStream class for detecting if there is any incoming data from the serial port. If there is, the response is logged. For testing with Arduino, the thread is set to wait 100 milliseconds before trying to read incoming data again, but this may not be the case when communicating with a satellite subsystem. The implementation of the thread can be seen on Figure 13.

```
receiveDataThread = new Thread(){
    @Override public void run() {
        // Wait after connecting, so the bootloader can finish.
        try {Thread.sleep(100); } catch(Exception e) {}

        LOG.debug("Ready to read serial port: " + serialPort.getSystemPortName());
        while(true) {
            Scanner data = new Scanner(serialPort.getInputStream());
            if(data.hasNextLine()){
                try {
                    String response = data.nextLine();
                    LOG.info("Response: " + response);
                    serialPortResponseProducer.sendBody(response);
                } catch (Exception e) {
                }
            }
            try {Thread.sleep(100); } catch(Exception e) {}
        }
    }
};
```

Figure 13: Thread used for receiving data from a serial port

5.4 Simulating subsystem communication with Arduino

For testing the module, a simple program has been written and uploaded to Arduino, shown on Figure 14, to get responses for the commands sent to it through the serial port. Arduino Software (IDE) [31] version 1.6.8 was used to write the program and upload it to the microcontroller. The program checks in a never-ending loop, if there are any incoming commands and if there are, it adds a random integer, 0 or 1, to the end of the command, and sends it back through the serial port. The random integer was used to get a Groovy test to pass and fail randomly.

```
1   char inputBuffer[10];
2   void setup() {
3     Serial.begin(9600);
4   }
5
6   void loop() {
7     if(Serial.available())
8     {
9       String text = Serial.readString();
10      int randomInt = random(0,2);
11      Serial.println(text + ";" + randomInt);
12    }
13  }
```

Figure 14: Code uploaded to Arduino microcontroller, used for testing

6 Example temperature-humidity sensor test

For a real-world example of how the testing module works, a simple temperature and humidity sensor was connected to the Arduino microcontroller, illustrated on Figure 15. The sensor model was DHT11 and DHTLIB library [32] was used to read the temperature, humidity, and status of the sensor.

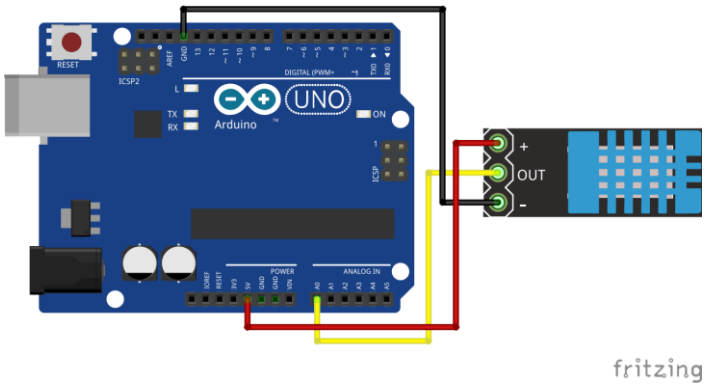

Figure 15: Arduino microcontroller (left) connected to a DHT11 sensor [33]

The test was set to fail based on the following criteria:

1. something wrong with the sensor, for example there is a connection or a checksum error;
2. temperature too high, more than 30℃;
3. humidity too high, more than 60%.

The example test was composed of three separate subtests, to clearly show which of the criteria were fulfilled, which were not and which were not run. An example can be seen on Figure 16, where the sensor was working correctly, so the test passed, but the temperature was higher (32.0 ℃) than what was set in the second criterion, so the test failed. Because the temperature test failed, and the humidity test was set to run after the temperature test, the humidity test was not run.

TEST RESULTS:

Name	Result	Details
StatusTest	PASSED	-
TemperatureTest	FAILED	Line: 28; Cause: Error, temperature is too high: 32.0
HumidityTest	-	-

Figure 16: Failed temperature test results

The code used for this test is in Appendix B and the code running on the Arduino, for getting sensor values, is in Appendix C.

7 Future developments

7.1 Future developments

Because the desired solution for satellite subsystem testing was not completed, there are plans to improve the existing system. Current plans involve fulfilling the three requirements described at the end of Chapter 8. Usability of the testing module can be improved as well. In addition, to be able to start testing subsystems, the hardware side of the testing system, including the protocol for software and hardware communication, must be completed, both of which fall out of the scope of this thesis.

7.1.1 MCS and testing module separation

One major improvement would be to separate the testing module from the MCS, so that the user can connect the subsystem they want to test to their personal computer and only run the testing module on their own computer. This is illustrated on Figure 17. The testing module would then communicate with the MCS over the Internet. This was taken into consideration when choosing TCP as the communication method between the MCS and testing module. At the moment, the user would have to run both the MCS and the testing module on their personal computer. This is not ideal, because it requires deep knowledge about the MCS to set it up on a stand-alone computer, which adversely impacts the usability of the testing module.

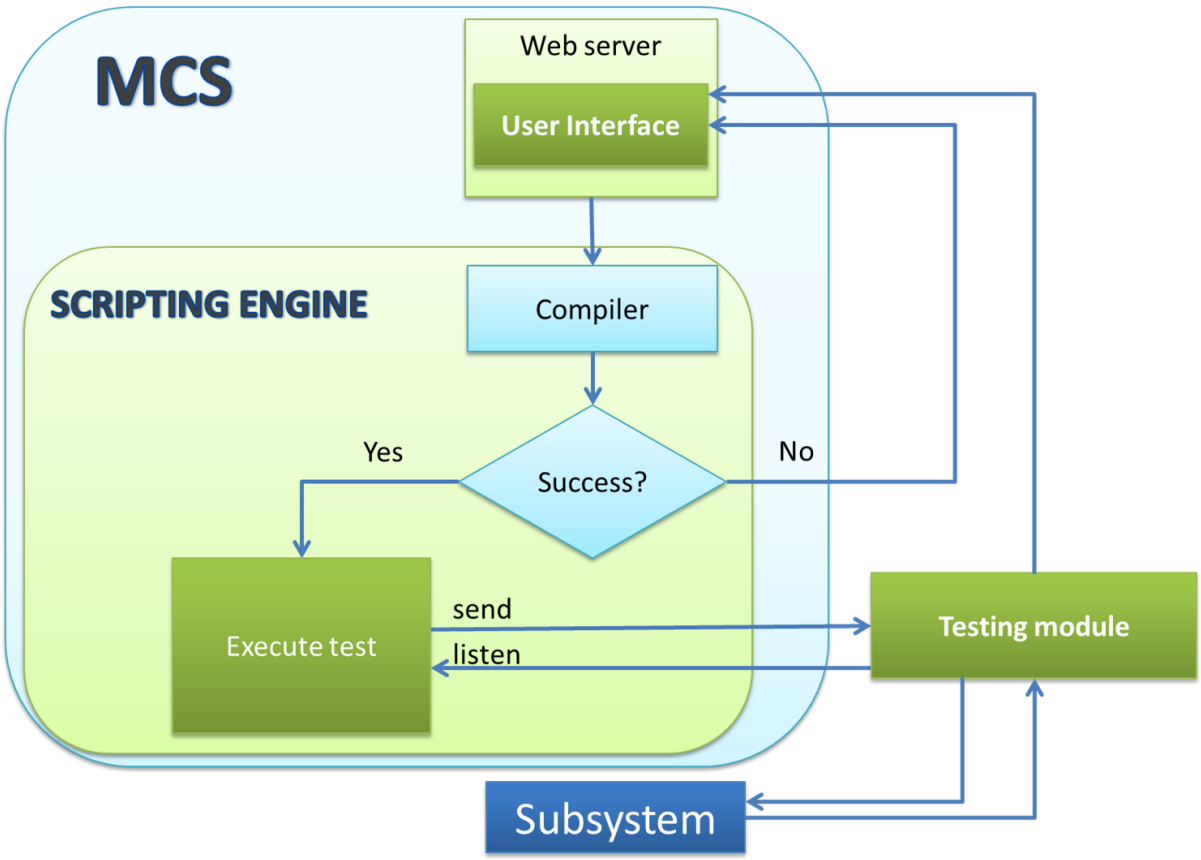

Figure 17: Data flow chart after separating testing module from the MCS

This solution comes with a couple of challenges. One of them would be the addition of Camel routes during runtime. The current Camel routes for communication between the MCS and the testing module can be seen on Figure 11. As can be seen from that code, the routes are configured for localhost. For this solution to work, new routes must be added during runtime, using the user's IP address.

The other problem that arises from this, is the need to add an identifier, that would be unique for each user, to every message sent in either Camel or through websockets. Otherwise, having multiple concurrent users would mean that all the users would receive each other's messages as well as their own. The unique identifier would be used to filter out any messages that were not intended for the current user, so that the testing would not be affected by other concurrent users.

8 Conclusion

This thesis presented the results of creating a first functional version of a testing module for satellite subsystem testing. The developed system allows to automate testing and creates test results and communication logs required for a more complete overview of the testing process. The module was successfully integrated with the MCS, so that the development of two separate systems would not be necessary.

The functionality of the developed module was verified by using a microcontroller and a simple temperature-humidity sensor. The sensor test in Chapter 6 emulated a realistic situation, where the EUT might have a restriction for maximum operating temperature, so temperatures higher than that indicate a problem with the EUT and the test should fail.

Three of the requirements listed in Chapter 3 were not fulfilled, because the integration of the testing module and the MCS took longer than expected. One of them was 1.6, which was partially fulfilled, because the user can start tests, but can't stop or pause them prematurely nor execute a test one step at a time. The others were 1.4 and 2.4, both of which concern the command history, with 1.4 being dependent on 2.4.

References

[1] A. J. Mroczek, "Determining the cost effectiveness of nano-satellites," p. 59, 2014.

[2] J. England-Nelson, "CubeSat miniature satellites poised to disrupt aerospace industry," 18.05.2014. [Online]. Available: http://www.dailybreeze.com/business/20140518/cubesat-miniature-satellites-poised-to-disrupt-aerospace-industry. [Accessed 12.05.2016].

[3] "ESTCube," [Online]. Available: http://to.ee/eng/research/research_topics/space_technology/estcube_team. [Accessed 12.05.2016].

[4] H. J. Kramer, "Aalto-1: The Finnish Student Nanosatellite," [Online]. Available: https://directory.eoportal.org/web/eoportal/satellite-missions/a/aalto-1. [Accessed 12.05.2016].

[5] "SwissCube Project," 05.06.2015. [Online]. Available: http://space.epfl.ch/page-39444-en.html. [Accessed 12.05.2016].

[6] M. Swartwout, "The First One Hundred CubeSats: A Statistical Look," *JOURNAL OF SMALL SATELLITES,* vol. 2, no. 2, p. 229, 2013.

[7] R. Munakata, W. Lan, A. Toorian, A. Hutputanasin and S. Lee, "CubeSat Design Specification Rev. 12," p. 5, 2009.

[8] R. Munakata, W. Lan, A. Toorian, A. Hutputanasin ja S. Lee, „CubeSat Design Specification Rev. 12," pp. 8-9, 2009.

[9] A. Mehrparvar, D. Pignatelli, J. Carnahan, R. Munakata, W. Lan, A. Toorian, A. Hutputanasin and S. Lee, "CubeSat Design Specification Rev. 13," p. 9, 2014.

[10] Hannes993, "The structure of cubesat ESTCube-1," 18.01.2013. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/7/77/The_structure_of_cubesat_ESTCube-1_eng.jpg. [Accessed 12.05.2016].

[11] H. J. Kramer, "ESTCube-1 (Estonian Student Satellite-1)," [Online]. Available: https://directory.eoportal.org/web/eoportal/satellite-missions/e/estcube-1. [Accessed 12.05.2016].

[12] "ESTCUBE-2," [Online]. Available: http://www.estcube.eu/en/estcube-2. [Accessed 12.05.2016].

[13] I. Iakubivskyi, H. Ehrpais, E. Ilbis, K. Reinkubjas, P. Janhunen, P. Toivanen, J. Envall and A. Slavinskis, "ESTCube-2 Mission Analysis: Plasma Brake Experiment for Deorbiting," 2016.

[14] "MCS," [Online]. Available: http://www.estcube.eu/en/satellite/mcs. [Accessed 12.05.2016].

[15] M. Doyle and J. Klug, "Hummingbird - A Service Based Open Source Ground Segment for Small Satellites," p. 1.

[16] ESTCube-1 Mission Control System team.

[17] "A multi-faceted language for the Java platform," [Online]. Available: http://groovy-lang.org/index.html. [Accessed 12.05.2016].

[18] "Program structure," [Online]. Available: http://groovy-lang.org/structure.html#_scripts_versus_classes. [Accessed 12.05.2016].

[19] "Elvior TestCast," 24.03.2016. [Online]. Available: http://www.verifysoft.com/en_elvior_testcast.html. [Accessed 12.05.2016].

[20] "TCP Definition," 19.10.2005. [Online]. Available: http://www.linfo.org/tcp.html. [Accessed

12.05.2016].

[21] “ArduinoUno R3 Front,” [Online]. Available: https://www.arduino.cc/en/uploads/Main/ArduinoUno_R3_Front_450px.jpg. [Accessed 12.05.2016].

[22] "Built for code," [Online]. Available: https://ace.c9.io/#nav=about. [Accessed 12.05.2016].

[23] "WebSockets," 09.05.2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Accessed 12.05.2016].

[24] "Data Binding," [Online]. Available: https://docs.angularjs.org/guide/databinding. [Accessed 12.05.2016].

[25] "How does a Queue compare to a Topic," [Online]. Available: http://activemq.apache.org/how-does-a-queue-compare-to-a-topic.html. [Accessed 12.05.2016].

[26] "Processor," [Online]. Available: http://camel.apache.org/processor.html. [Accessed 12.05.2016].

[27] "Class GroovyShell," [Online]. Available: http://docs.groovy-lang.org/latest/html/api/groovy/lang/GroovyShell.html. [Accessed 12.05.2016].

[28] "Platform-independent serial port access for Java," [Online]. Available: http://fazecast.github.io/jSerialComm/. [Accessed 12.05.2016].

[29] "Apache log4j™ 1.2," [Online]. Available: https://logging.apache.org/log4j/1.2/. [Accessed 12.05.2016].

[30] J. Jenkov, "BlockingQueue," 23.06.2014. [Online]. Available: http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html. [Accessed 12.05.2016].

[31] "Download the Arduino Software," [Online]. Available: https://www.arduino.cc/en/Main/Software. [Accessed 12.05.2016].

[32] "DHT11-Humidity-TempSensor," [Online]. Available: https://arduino-info.wikispaces.com/DHT11-Humidity-TempSensor. [Accessed 12.05.2016].

[33] "Schematic," [Online]. Available: https://brainy-bits.com/wp-content/uploads/2014/12/schematic-dht11t.png. [Accessed 12.05.2016].

[34] A. Gpredict, 03.12.2009. [Online]. Available: http://gpredict.oz9aec.net/. [Accessed 12.05.2016].

Appendix A: sample ESTCube-1 contact times

Below is the list of contact times of ESTCube-1 from the 18th to 20th of April 2016. The data is generated by GPredict [34], which is a satellite tracking application.

Observer: T2he, Tartu/Ulenurme, Estonia
LAT:58,37 LON:26,73

AOS	TCA	LOS	Duration	Max El	AOS Az	LOS Az
2016/04/18 22:57:49	2016/04/18 23:04:26	2016/04/18 23:11:02	00:13:13	56,49	147,88	345,70
2016/04/19 00:35:10	2016/04/19 00:41:29	2016/04/19 00:47:49	00:12:38	27,32	195,93	338,77
2016/04/19 02:16:20	2016/04/19 02:19:49	2016/04/19 02:23:18	00:06:57	3,51	256,87	321,19
2016/04/19 10:41:22	2016/04/19 10:45:39	2016/04/19 10:49:57	00:08:35	5,88	34,02	115,18
2016/04/19 12:17:16	2016/04/19 12:23:47	2016/04/19 12:30:17	00:13:01	35,14	19,83	171,91
2016/04/19 13:54:07	2016/04/19 14:00:40	2016/04/19 14:07:12	00:13:05	44,36	13,57	219,16
2016/04/19 15:31:10	2016/04/19 15:36:26	2016/04/19 15:41:42	00:10:31	12,20	10,19	265,41
2016/04/19 17:08:04	2016/04/19 17:11:17	2016/04/19 17:14:29	00:06:25	3,24	10,21	311,63
2016/04/19 18:43:18	2016/04/19 18:45:36	2016/04/19 18:47:54	00:04:35	1,54	25,79	344,71
2016/04/19 20:15:58	2016/04/19 20:20:05	2016/04/19 20:24:13	00:08:14	5,93	67,90	350,44
2016/04/19 21:49:22	2016/04/19 21:55:18	2016/04/19 22:01:14	00:11:51	20,43	114,59	348,64
2016/04/19 23:24:51	2016/04/19 23:31:33	2016/04/19 23:38:14	00:13:22	88,39	160,94	344,18
2016/04/20 01:03:03	2016/04/20 01:08:56	2016/04/20 01:14:50	00:11:47	17,56	210,70	335,73
2016/04/20 11:08:03	2016/04/20 11:13:19	2016/04/20 11:18:36	00:10:32	10,88	28,28	133,24
2016/04/20 12:44:24	2016/04/20 12:51:05	2016/04/20 12:57:46	00:13:22	56,61	17,66	185,54
2016/04/20 14:21:20	2016/04/20 14:27:38	2016/04/20 14:33:56	00:12:36	29,65	12,38	232,06
2016/04/20 15:58:23	2016/04/20 16:03:08	2016/04/20 16:07:52	00:09:28	8,67	9,72	278,46
2016/04/20 17:35:05	2016/04/20 17:37:46	2016/04/20 17:40:27	00:05:22	2,16	11,82	323,46
2016/04/20 19:09:24	2016/04/20 19:12:04	2016/04/20 19:14:44	00:05:19	2,12	36,04	348,02

Appendix B: example Groovy test code

```
1   import static org.junit.Assert.assertEquals
2   import static org.junit.Assert.assertTrue
3
4   state(StatusTest) {
5       log "Sensor status test."
6
7       Map<String, Object> expectedResponse = new HashMap<String, Object>();
8       expectedResponse.put("response", "OK");
9
10      send("sensorStatus")
11      def sensorStatus = listen("sensor")
12      log "Received $sensorStatus"
13
14      String sensorStatusF = temp.get("response")
15      //Test if expected response equals actual response
16      assertEquals("OK", sensorStatusF)
17  }
18
19  state(TemperatureTest) {
20      log "Temperature test."
21
22      send("temperature")
23      Map<String, String> temp = listen("sensor")
24      log "Received $temp"
25
26      float tempF = Float.parseFloat(temp.get("response"))
27
28      assertTrue("Error, temperature is too high: $tempF", 30 >= tempF)
29  }
30
31  state(HumidityTest) {
32      log "Temperature test."
33
34      send("humidity")
35      Map<String, String> humidity = listen("sensor")
36      log "Received $humidity"
37
38      float humidityF = Float.parseFloat(humidity.get("response"))
39
40      assertTrue("Error, humidity is too high: $humidityF", 50 >= humidityF)
41  }
42
43  while(true){
44      gotoState StatusTest
45      gotoState TemperatureTest
46      gotoState HumidityTest
47  }
```

Appendix C: Arduino code for temperature-humidity sensor

```
#include <dht.h>
/* These are function prototypes - required by the compiler */
void humidity();
void temperature();
void sensorStatus();

typedef void (* Caller)();
//Initialize addresses for pointers
Caller FuncCall[] = {&humidity, &temperature, &sensorStatus};
String funcList[] = {"humidity", "temperature", "sensorStatus"};

dht DHT;

#define DHT11_PIN 7

void setup()
{
  Serial.begin(115200);
}

void loop()
{
  //If a command is received, execute the corresponding method
  if(Serial.available())
  {
        String command = Serial.readString();
        int arraySize = sizeof(funcList)/sizeof(funcList[0]);
        int commandFound = 0;
        for (int i = 0; i < arraySize; i++) {
          if(command == funcList[i]){
            commandFound = 1;
            FuncCall[i]();
            break;
          }
        }
        if(commandFound != 1){
          Serial.println("Command not recognized.");
        }
  }
}

void temperature(){
  Serial.println(DHT.temperature, 1);
}
void humidity(){
  Serial.println(DHT.humidity, 1);
}
void sensorStatus(){
  int chk = DHT.read11(DHT11_PIN);
  switch (chk)
  {
    case DHTLIB_OK:
      Serial.println("OK");
      break;
    case DHTLIB_ERROR_CHECKSUM:
      Serial.println("Checksum error");
      break;
    case DHTLIB_ERROR_TIMEOUT:
      Serial.println("Time out error");
      break;
    case DHTLIB_ERROR_CONNECT:
      Serial.println("Connect error");
      break;
    case DHTLIB_ERROR_ACK_L:
      Serial.println("Ack Low error");
      break;
    case DHTLIB_ERROR_ACK_H:
      Serial.println("Ack High error");
      break;
    default:
      Serial.println("Unknown error");
      break;
  }
}
```

Appendix D: repository

The URL for the repository containing the modified and added MCS modules: https://github.com/aivarl/HIL-Testing-Module-for-MCS.

Appendix E: license

Non-exclusive licence to reproduce thesis and make thesis public

I, Aivar Lobjakas,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Hardware-in-the-loop testing module for Mission Control System,

supervised by Kaarel Hanson, Indrek Sünter.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 12.05.2016